Security Review Report
NM-0289 EtherFi Cash

NETHERMIND
SECURITY

(Sep 23, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for the EtherFi Cash. This security engagement focused on the smart contracts that constitute part of the new payment system developed by the **EtherFi**. The **EtherFi Cash** payment system is an ambitious project that aims to remove the friction between web3 DeFi and traditional financial systems. It allows `EtherFi Cash` users (after performing the KYC procedure) to create a `UserSafe` contract in which they can manage on-chain assets. The funds deposited to `UserSafe` can then be spent with `EtherFi Card` in off-chain services. `UserSafe` lets users set spending limits, set up an account recovery mechanism, swap with external protocols, and lend tokens from EtherFi's `L2DebtManager`.

During the audit, the client changed the initial project in several commits. **The first change** added important changes and features at the commit c261affeee414beb2a7e924d26a0432328e0ea22:

- setting separate LTV and Liquidation Threshold for each collateral token
- setting the separate interest rate for each debt token
- permissionless supply of the borrow tokens, which grants supplies shares that accrue in value
- permissionless liquidation
- closing account restriction to only debt-free accounts

The description in the Executive Summary and System Overview includes the changes described above.

Second change includes modification of the `UserSafeFactory` in commit ec447998ac6bd10ea97f92fc9f872faccb513cb5 introduces `CREATE3` mechanism to deploy new `UserSafe` contracts.

The last important change is refactoring the `L2DebtManager` by splitting it in four contracts: `DebtManagerAdmin`, `DebtManagerCore`, `DebtManagerInitializer`, `DebtManagerStorage`. The `L2DebtManager` is substituted by the four contracts with no code difference except split to different files, and therefore it is removed at the commit d135e5a245b82075fdbde5b292eaa5953862b546.

The audit scope does not cover interactions with any specific external swap protocols in `UserSafe.swapAndTransfer(...)` and interactions with AAVE in `L2DebtManager.fundManagementOperation(...)`.

This audit does not cover analysis of the initial parameters settings.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** thirty-seven points of attention, where five are classified as `Critical`, two are classified as `High`, eleven are classified as `Medium`, seven are classified as `Low`, and thirteen are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

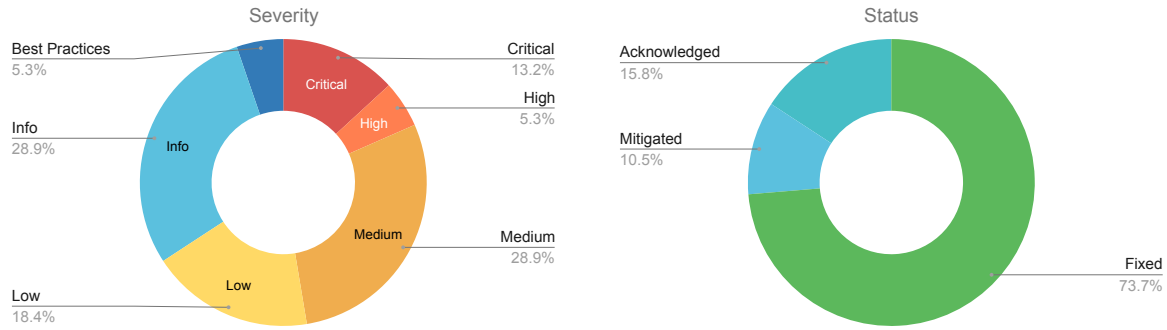

Fig. 1: Distribution of issues: Critical (5), **High** (2), **Medium** (11), **Low** (7), **Undetermined** (0), **Informational** (11), **Best Practices** (2). **Distribution of status: Fixed** (25), **Acknowledged** (6), **Mitigated** (4), **Unresolved** (3)

Summary of the Audit

Audit Type	Security Review
Initial Report	Sep 18, 2024
Response from Client	Regular responses during audit engagement
Final Report	Sep 23 2024
Repository	cash-contracts
Commit (Audit)	b15c678c85d4c5a335f57ae217c8f48b0131a18a
Commit (Final)	867b3b4c66ea715079700b9cf07891c13089d852
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	UserSafeFactory.sol	15	11	73%	5	31
2	UserSafe.sol	568	99	17%	106	773
3	L2DebtManager.sol	866	168	19%	186	1220
4	SignatureUtils.sol	31	4	12%	5	40
	Total	**1480**	**282**	**19.1%**	**302**	**2064**

3 Summary of Issues

	Finding	Severity	Update
1	Collateral tokens can be drained with `withdrawBorrowToken(...)`	Critical	Fixed
2	Incorrect calculation of borrow amount	Critical	Fixed
3	Incorrect decimals handling in `supply(...)`	Critical	Fixed
4	Incorrect share calculation in `supply(...)`	Critical	Fixed
5	Inflation attack	Critical	Fixed
6	Funds can be borrowed without the debt	High	Fixed
7	Lack of incentive for liquidators	High	Fixed
8	Features that are planned to add require substantial changes in the code	Medium	Acknowledged
9	Incorrect calculation in `_getAmountWithInterest(...)`	Medium	Mitigated
10	Lack of collateral cap	Medium	Fixed
11	Lack of collateral control on liquidation	Medium	Fixed
12	Liquidations can be blocked	Medium	Fixed
13	Optional partial liquidations	Medium	Fixed
14	Price Oracle may not support collateral assets	Medium	Fixed
15	Unhandled bad debt	Medium	Mitigated
16	Unsupporting tokens may be practically impossible	Medium	Acknowledged
17	Use of deprecated Chainlink Function leading to insufficient data feed validation	Medium	Mitigated
18	`L2DebtManger` is permissionless	Medium	Fixed
19	Updating borrowings with collateral token	Low	Fixed
20	Any tokens can be supplied as borrow tokens	Low	Fixed
21	Lack of `disableInitializers()`	Low	Fixed
22	Lack of minimum borrow amounts	Low	Acknowledged
23	Not following the CEI pattern	Low	Fixed
24	The `_requestWithdrawal(...)` function does not check for duplicates	Low	Acknowledged
25	The recovery mechanism does not protect the Safe owner	Low	Fixed
26	No distinction between the KYC and non-KYC `UserSafe` contracts	Info	Fixed
27	No checks of liquidation threshold compared to liquidation bonus	Info	Fixed
28	Non-USD stablecoins can't be supported	Info	Mitigated
29	The `CREATE3` library can't be used on zkSync Era	Info	Acknowledged
30	The `accInterestAlreadyStored` variable is unused	Info	Fixed
31	The `onlyOwner` is usable only if the owner is an ETH address	Info	Fixed
32	The recovery pubkeys may only be ETH addresses	Info	Acknowledged
33	Unclear result in `debtRatioOf(...)`	Info	Fixed
34	Unnecessary owner in the recovery mechanism	Info	Fixed
35	Unused Code	Info	Fixed
36	Unused comparison expressions	Info	Fixed
37	Improve code clarity and readability by using constants instead of magic numbers.	Best Practices	Fixed
38	Use AccessControlDefaultAdminRulesUpgradeable	Best Practices	Fixed

4 System Overview

The main entry point for the KYC'ed users is `UserSafe`, to which users deposit on-chain funds and with which users interact to manage funds. The `UserSafe` can transfer funds to the multisig wallet controlled by EtherFi, which is connected to the off-chain card payment infrastructure. The `UserSafe` contracts are created from a factory contract `UserSafeFactory`. `UserSafe` can perform swaps with external protocols and use `L2DebtManager` to take over-collateralized loans. When the user transfers funds calling `UserSafe.transfer(...)`, `UserSafe.swapAndTransfer(...)` or takes a loan with `UserSafe.borrow(...)`, the funds are sent to the `EtherFi Multisig` and are considered as spent. Those funds are used to pay with a card. The `L2DebtManager` calculates the value of collateral by calling `PriceProvider`, which fetches the price from the `Chainlink Oracle`. The USD stablecoins comprise the borrowing pool in `L2DebtManager`. Those funds can be supplied by the non-KYC'ed participants who get shares in exchange. Those shares accrue in value over time by increasing the borrowers' loan value. The unhealthy loans can be liquidated by non-KYC liquidators who are incentivized by the liquidation bonus.

The following sections delve into the system's components and their interactions. The diagram below showcases a high-level view of the system's architecture.

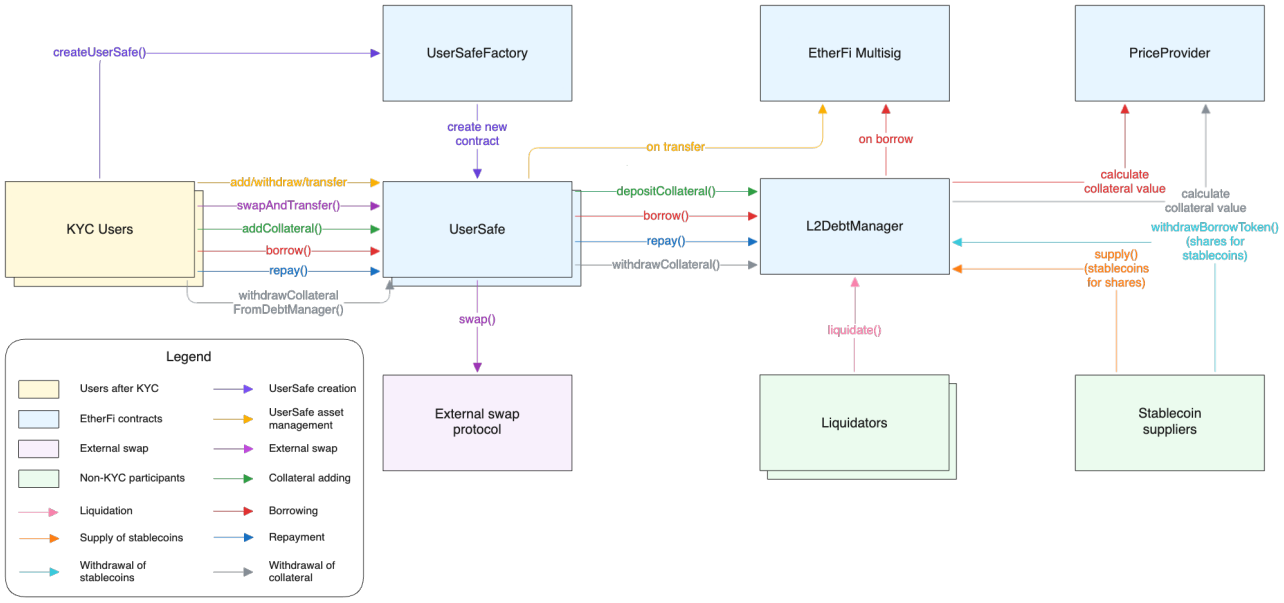

Fig. 2: EtherFi Cash overview of smart contracts. The diagram presents only the core flows, excluding the standard configuration and admin-specific actions.

4.1 Protocol participants

The `EtherFi Cash` has following participants:

- **Users** - the protocol users that went through the KYC (Know Your Customer) process and are eligible to create `UserSafe` contract, which can be utilized in the card payment process. Users are the owners of the `UserSafe`
- **EtherFi Cash Wallet** - special address controlled by the EtherFi. From this address, users can call their `UserSafe` contracts to manage funds
- **EtherFi Cash Multisig** - special address controlled by the EtherFi, which receives funds on transfers and borrows. Funds in this address can be used to perform card payments
- **Contracts Owner** - special addresses controlled by the EtherFi that control the `L2DebtManager` and `UserSafeFactory` contracts
- **Borrow tokens suppliers** - any address can supply USD stablecoins to the `L2DebtManager` in exchange of shares
- **Liquidators** - any address can perform liquidations of the liquidatable loans

4.2 UserSafe - funds management

The users, after the KYC process, are eligible to register the `UserSafe` contract so that funds from this contract can be used to spend with a `EtherFi Card`. The user who is the owner of the `UserSafe` contract can perform configuration actions on the `UserSafe` contract with the following functions:

- `setOwner(...)` - checks the provided signature and sets new owner for the `UserSafe` contract
- `resetSpendingLimit(...)` - sets spending limit with emptying the spent amount

- `updateSpendingLimit(...)` - changes spending limit while preserving the spent amount
- `setCollateralLimit(...)` - sets the limit for deposited collateral
- `requestWithdrawal(...)` - creates a request withdrawal which defines assets and amounts that will be withdrawn after the predefined period.
- `processWithdrawal(...)` - permisionless function that finalizes the withdrawal request process

All the actions that involve moving funds to or out of the `UserSafe` must be performed from the wallet controlled by the EtherFi. Functions that allow performing those actions are listed below:

- `transfer(...)` - transfers funds form the `UserSafe` to the `EtherFiCashMultiSig` address. Funds transferred to the multisig wallet are considered to be spent by the user and can be utilized with `EtherFi Card` payments
- `swapAndTransfer(...)` - performs a swap with external swap protocol. The received amount is transferred to the `EtherFiCashMultiSig` address and can only be utilized with `EtherFi Card` payments
- `addCollateral(...)` - transfers funds out from the `UserSafe` and deposits them to the `L2DebtManager` as a collateral
- `borrow(...)` - calls `L2DebtManager.borrow(...)` and creates loan. The borrowed assets are immediately transferred to the `EtherFiCashMultiSig` and can only be utilized with `EtherFi Card` payments
- `addCollateralAndBorrow(...)` - performs deposit of collateral and borrowing funds in the same transaction
- `repay(...)` - repays the debt stored in `L2DebtManager`
- `withdrawCollateralFromDebtManager(...)` - withdraws the collateral from the `L2DebtManager` and transfers it to the `UserSafe`
- `closeAccountWithDebtManager(...)` - closes account in `L2DebtManager(...)` if possible and transfers funds to the `UserSafe`

The `recoverUserSafe(...)` function is permissionless but requires two valid signatures from predefined recovery addresses. This function allows for emergency recovery of the `UserSafe` account by changing the current owner to the new owner.

4.3 L2DebtManager - loans mechanism

The loan mechanism allows users to borrow supported USD stablecoins pooled in the `L2DebManager`. Total debt and total collateral are tracked for each user, so it is not possible for a single user to create more than one loan. To borrow a stablecoin, the user needs to provide enough collateral since the loans are overcollateralized. The collateral may be provided only in the form of supported tokens with the `depositCollateral(...)` function. The amount of collateral is stored in native value and transformed to value in USD (by current price) when needed. The collateral may be withdrawn only if the state of the loan remains healthy after the withdrawal. The health of a loan is defined for each token in the `_ltv` mapping, which represents the maximum proportion of the debt to collateral under which the loan is healthy. The borrowing tokens can be supplied by any address in exchange for shares. Shares accrue in value over time, which incentivizes the suppliers to provide liquidity to the `L2DebtManager`. The borrowers are obligated to repay a higher amount of debt, which constitutes the supplier's rewards. The loan can become a liquidatable state, which is defined for each token in the `_liquidationThreshold` mapping, which represents the proportion of the debt to collateral above which the loan is considered liquidatable. Liquidation is permissionless, and liquidators are incentivized by the reward in the form of an additional collateral bonus defined for each token. Below, we list core user-facing functions of `L2DebManager`:

- `supply(...)` - transfers the stablecoin from the caller, calculates the amount shares, which represent the portion of all the amount of given tokens and stores the number of shares
- `withdrawBorrowToken(...)` - calculates the number of shares that need to be removed for a given token amount, updates shares, and transfers the stablecoin to the caller
- `depositCollateral(...)` - transfers the collateral token from the caller and stores the amount
- `withdrawCollateral(...)` - updates stored collateral, checks the health status of the loan, and transfers collateral token to the caller
- `borrow(...)` - updates user's debt, checks loan's health, and transfers the stablecoin to the caller
- `repay(...)` - updates user's debt and transfers funds from the user
- `closeAccount(...)` - checks if the user does not have any debt and transfers to the caller all the collateral
- `liquidate(...)` - checks if the loan is liquidatable, transfers stablecoin from the user, calculates the amount of the collateral needed (with liquidation bonus), updates user's debt and collateral, sends the collateral to the liquidator

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Collateral tokens can be drained with `withdrawBorrowToken(...)`

File(s): L2DebtManager

Description: The `withdrawBorrowToken(...)` function does not validate if the provided token is a borrow token. This allows the malicious user to call the `withdrawBorrowToken(...)` function with a collateral token. Additionally, the function successfully transfers the tokens to the caller even if the number of subtracted shares is zero:

```
function withdrawBorrowToken(address borrowToken, uint256 amount) external {
    // @audit: no checks if the token is a borrow token
    uint256 totalBorrowTokenAmt = _getTotalBorrowTokenAmount(borrowToken);
    if (totalBorrowTokenAmt == 0) revert ZeroTotalBorrowTokens();
    // @audit: any amount of the requested withdrawal token
    //         would result in zero shares since
    //         the totalSharesOfBorrowTokens is empty
    //         for any collateral token
    uint256 shares = (amount *
        _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens) /
        totalBorrowTokenAmt;

    if (_sharesOfBorrowTokens[msg.sender][borrowToken] < shares)
        revert InsufficientBorrowShares();

    _sharesOfBorrowTokens[msg.sender][borrowToken] -= shares;
    _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens -= shares;
    // @audit: the requested amount is sent to the attacker
    IERC20(borrowToken).safeTransfer(msg.sender, amount);
    emit WithdrawBorrowToken(msg.sender, borrowToken, amount);
}
```

Consider the following scenario:

- user A deposits 100 weETH as a collateral;
- malicious user calls `withdrawBorrowToken(...)` function with `borrowToken = weETH` and `amount = 100 weETH`;

- calculated shares are zero since there are no registered shares for `weETH` in `_borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens`, therefore, it's empty. The shares calculation is defined with: - `amount * _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens / totalBorrowTokenAmt` - would result in: - `100 * 0 / 100 = 0` - the check `if (_sharesOfBorrowTokens[msg.sender][borrowToken] < shares)` would pass, since `0 < 0 = false` and the revert won't be reached

As a result, the malicious user is able to transfer any amount out of the debt manager.

Recommendation(s): Consider checking if the `token` parameter is the address of the supported borrow token. Additionally, check if the calculated number of shares is `0`.

Status: Fixed

Update from the client: Did not add `isBorrowToken` check because if we remove support for a borrow token, users should still be able to withdraw. Added a check that calculated shares should not be equal to `0`.

6.2 [Critical] Incorrect calculation of borrow amount

File(s): L2DebtManager.sol

Description: The borrowed amount is calculated differently during borrowing and repayment, which leads to repaying debt in much less than the amount borrowed. The `borrow(...)` function stores debt by scaling the borrowed token `amount` to `6` decimals, and send the user original `amount`. However, the `repay(...)` function does the opposite, it decreases debt by the provided `amount` and transfers from the user the `amount` scaled to `6` decimals:

```
// BORROW
function borrow(
    address token,
    uint256 amount
) external updateBorrowings(msg.sender) {
    if (!isBorrowToken(token)) revert UnsupportedBorrowToken();

    uint256 borrowAmt = _convertToSixDecimals(token, amount);
    // @audit: increase the debt by amount scaled down to 1e18
    _userBorrowings[msg.sender] += borrowAmt;
    _totalBorrowingAmount += borrowAmt;

    if (debtRatioOf(msg.sender) > _liquidationThreshold)
        revert InsufficientCollateral();

    if (IERC20(token).balanceOf(address(this)) < amount)
        revert InsufficientLiquidity();
    // @audit: send the unscaled amount to user
    IERC20(token).safeTransfer(
        _cashDataProvider.etherFiCashMultiSig(),
        amount
    );

    emit Borrowed(msg.sender, token, amount);
}
// REPAY
function _repayWithBorrowToken(
    address token,
    address user,
    uint256 repayDebtUsdcAmt
) internal {
    IERC20(token).safeTransferFrom(
        msg.sender,
        address(this),
        // @audit: transfer from user amount scaled down
        _convertToSixDecimals(token, repayDebtUsdcAmt)
    );
    // @audit: decrease debt by unscaled amount
    _userBorrowings[user] -= repayDebtUsdcAmt;
    _totalBorrowingAmount -= repayDebtUsdcAmt;

    emit RepaidWithUSDC(user, msg.sender, repayDebtUsdcAmt);
}
```

In effect, users can borrow tokens and repay all the debt with a fraction of what was borrowed. This may lead to stealing funds from the debt manager. Note that this issue exists only if the borrowed token has more than `6` decimals.

Recommendation(s): Consider scaling amount during the repay the same way as on borrow.

Status: Fixed

Update from the Nethermind: Resolved by changes in commit `d97d9a9`.

6.3 [Critical] Incorrect decimals handling in `supply(...)`

File(s): `L2DebtManager.sol`

Description: The `supply(...)` function does not correctly calculate the shares in certain conditions. The supply calculation is presented below:

```
function supply(
    address user,
    address borrowToken,
    uint256 amount
) external {
    if (!isBorrowToken(borrowToken)) revert UnsupportedBorrowToken();
    uint256 shares = _borrowTokenConfig[borrowToken]
        .totalSharesOfBorrowTokens == 0
        ? amount
        : amount.mulDiv(
            _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens,
            _getTotalBorrowTokenAmount(borrowToken),
            Math.Rounding.Floor
        );
    ...
}

function _getTotalBorrowTokenAmount(
    address borrowToken
) internal view returns (uint256) {
    return
        // @audit: result of this function is always in 6 decimals
        totalBorrowingAmount(borrowToken) +
        // @audit: result of this function can be other than 6 decimals
        IERC20(borrowToken).balanceOf(address(this));
}
```

The amount of all borrowing tokens is returned by the `_getTotalBorrowTokenAmount(...)` function. However, this function may return an incorrect result if the `borrowToken` has different decimals than `6` since the result could be: `amount in 6 decimals + amount in 18 decimals`. This leads to a decrease in the `_getTotalBorrowTokenAmount(...)` on every borrow of the `borrowToken` in incorrectly calculated shares. Consider the following scenario:

- user A supplies `10 * 1e18` of DAI and receives `10 * 1e18` shares (first supplier);
- user B borrows all `10 * 1e18` DAI, and debt in `_borrowTokenConfig[token].totalBorrowingAmount` is registered as `10 * 1e6`;
- user B supplies `1e18` of DAI, the calculation of shares:;

```
1e18 * 10e18 / (10 * 1e6) = 1e30
```

The above calculation is incorrect and allows malicious users to mint many more shares than they should. The correct calculation should mint user `1e18` shares in the presented scenario.

Recommendation(s): Consider scaling the output of `totalBorrowingAmount(borrowToken)` to the decimals of the `borrowToken` in the `_getTotalBorrowTokenAmount(...)`.

Status: Fixed

6.4 [Critical] Incorrect share calculation in `supply(...)`

File(s): `L2DebtManager.sol`

Description: **File(s)**: `L2DebtManager.sol`

Description: The calculation of the shares is done incorrectly for tokens with decimals other than `1e18` due to scaling by `1e18` in `_convertBorrowToShare(...)`. The shares in the `supply(...)` function are calculated by `assets amount * total shares / total assets`:

```
amount *
_borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens) /
_getTotalBorrowTokenAmount(borrowToken)
```

While this calculation is correct, the result (amount of shares) is passed to the `_convertBorrowToShare(...)` function:

```
function _convertBorrowToShare(
    address borrowToken,
    uint256 amount
) internal view returns (uint256) {
    // @audit: ONE_SHARE = 1e18
    return (amount * ONE_SHARE) / 10 ** _getDecimals(borrowToken);
}
```

The resulting shares are scaled to `1e18` decimals, and this value is stored as supplier's shares:

```
_sharesOfBorrowTokens[user][borrowToken] += shares;
_borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens += shares;
```

This scaling is incorrect as now, during the next supply, the shares will be calculated as: `assets amount * total shares * 1e18 / total assets` Consider following example on the `USDC`:

- user A supplies 1 USDC (1e6):;

- calculated shares: `1e6` - scaled shares: `1e6 * ONE_SHARE / 1e6 = 1e18` - totalSharesOfBorrowTokens = `1e18` - _sharesOfBorrowTokens = `1e18`

- user B supplies 1 USDC:;

- calculated shares: `1e6 * 1e18 / 1e6 = 1e18` - scaled shares: `1e18 * ONE_SHARE / 1e6 = 1e30` - totalSharesOfBorrowTokens = `1e18 + 1e30` - _sharesOfBorrowTokens = `1e30`

As shown in the example, each next supplier receives `1e12` times more shares than they should receive. This behavior results in a few problems:

- each next supplier is granted much more shares than should which breaks the token distribution, as the last supplier have so much shares that can withdraw all the supplied tokens;
- after a few supplies, the `totalSharesOfBorrowTokens` overflows, and no more supplies are possible to perform;
- scaling is not taken into account during token withdrawal;

Note that this problem does not apply to the tokens with `1e18` decimals.

Recommendation(s): Consider removing the scaling of shares by `1e18`.

Status: Fixed

6.5 [Critical] Inflation attack

File(s): `L2DebtManager.sol`

Description: During the first deposit, the attacker may manipulate the calculation of the share, which would result in stealing the first depositor's funds. The borrowing assets may be provided with the `supply(...)` function, which mints the supplier shares. The amount of minted shares is equal to the deposited amount if there are no previously minted shares, and for any subsequent supplies it is calculated with:

```
depositedAmount * totalSharesOfBorrowTokens /
_getTotalBorrowTokenAmount(...)
```

The `_getTotalBorrowTokenAmount` compromises of the `totalBorrowingAmount + borrowToken` balance of the contract

```
    function _getTotalBorrowTokenAmount(
        address borrowToken
    ) internal view returns (uint256) {
        return
            totalBorrowingAmount(borrowToken) +
            // @audit: the balance of the contract
            //         may be manipulated without
            //         minting shares by doing
            //         direct token transfers
            IERC20(borrowToken).balanceOf(address(this));
    }
```

Since the `_getTotalBorrowTokenAmount(...)` includes the token balance of the `L2DebtManager`, the output of this function can be manipulated by the direct token transfer to the `L2DebtManager`. Because of that, the attacker may call `supply(...)` minting one share, and when the valid depositor wants to supply, the attacker front-runs the second user deposit by direct token transfer of an amount higher than the valid supplier. Such action results in minting zero shares for the supplier and granting deposited funds to the attacker. Consider following scenario:

1. User 1 supplies 1 BorrowToken to the contract;
2. User 2, the victim, wants to supply 200 BorrowTokens in the contract, but User 1 front-runs this transaction and directly sends 200 BorrowTokens to the Contract;
3. This inflates the `_getTotalBorrowTokenAmount ()` as borrowTokenBalance of the contract is increased to 201 BorrowTokens;
4. When User2 supplies the 200 BorrowTokens, they get 0 shares as `(1 * 200)/201)) == 0 shares`;
5. User 1, the attacker then withdraws 401 BorrowTokens because their one share represents the entire contract balance as the `totalSharesOfBorrowTokens` is;

Recommendation(s): Consider mitigating the described attack by tracking the amount of deposited assets in state variables, which would disallow share amount manipulation by direct transfers. Other alternative mitigations can be found in this article.

Status: Fixed

Update from the client: Added minSharesToMint for each borrow token.

Update from Nethermind Security: The changes applied in 1d4c37b try to handle the issue of inflation attack by adding an `minSharestToMint` check which allows to minit a minimum amount of shares:

```
        if (shares < _borrowTokenConfig[borrowToken].minSharesToMint)
                    revert SharesCannotBeLessThanMinSharesToMint();
```

Furthermore, the `withdrawBorrowToken(...)` function does not allow zero shares withdrawal. However, an attacker is still able to bypass the `minSharesToMint` check by supplying a huge amount of tokens and get shares such that the shares minted is above the minimum threshhold and then immediately call `withdrawBorrowToken (..)` leaving a tiny amount of shares in the contract, effecitvely bypassing the check. Contract in this state is still vulnerable to an inflation attack. Consider applying solutions discussed in this article. Alternatively, consider introducing the check for minimum amount of shares also in the `withdrawBorrowToken(...)` function.

Update from the client: Additional check in the `withdrawBorrowToken(...)` function was added to the `DebtManagerCore` at commmit 71db6c5f75de0d9070a62fe67f78c844bb9170f2.

6.6 [High] Funds can be borrowed without the debt

File(s): `L2DebtManager.sol`

Description: The rounding error during the calculation of the `borrowAmt` may be exploited to borrow funds without registering it. The `borrow(...)` function takes the `amount` parameter, which is first converted to `6` decimals with the `_convertToSixDecimals(...)` function, and the result is added to the `_userBorrowings` and `_userBorrowingAmount` mappings. Next, the `amount` is transferred to the borrower. However, the rounding error in `_convertToSixDecimals(...)` may cause the registered debt to be actually `0` if the borrowed `amount` is considerably small.

```
function _convertToSixDecimals(
    address token,
    uint256 amount
) internal view returns (uint256) {
    uint8 tokenDecimals = _getDecimals(token);
    // @audit: rounding error may output 0 for a non-zero amount
    return
        tokenDecimals == 6 ? amount :
        (amount * 1e6) / 10 ** tokenDecimals;
}

function borrow(
    address token,
    uint256 amount
) external updateBorrowings(msg.sender) {
    if (!isBorrowToken(token)) revert UnsupportedBorrowToken();

    // @audit: borrowAmt may be rounded to 0
    uint256 borrowAmt = _convertToSixDecimals(token, amount);
    // @audit: debt would be increased by 0
    _userBorrowings[msg.sender] += borrowAmt;
    _totalBorrowingAmount += borrowAmt;

    if (debtRatioOf(msg.sender) > _liquidationThreshold)
        revert InsufficientCollateral();

    if (IERC20(token).balanceOf(address(this)) < amount)
        revert InsufficientLiquidity();
    // @audit: even if debt would be increased by 0
    //         the original amount is sent to the borrower
    IERC20(token).safeTransfer(
        _cashDataProvider.etherFiCashMultiSig(),
        amount
    );

    emit Borrowed(msg.sender, token, amount);
}
```

Consider following scenario:

- user A calls `borrow(...)` with `amount=10` **11** of `DAI` (`18` decimals);
- `_convertToSixDecimals(...)` calculates:;

`(10` **`11 * 1e6) / 10`**`18 = 1e17 / 1e18 = 0`

- `_userBorrowings` and `_totalBorrowingAmount` are increased by `0`;
- user A receives `1e11` DAI;

Recommendation(s): Consider mitigating such a scenario by checking if the calculated debt is zero and rounding in favor of the protocol during division.

Status: Fixed

Update from the client:

6.7 [High] Lack of incentive for liquidators

File(s): L2DebtManager

Description: To protect the `L2DebtManager` from bad debt, the liquidation mechanism should efficiently remove loans that are not healthy. Currently, the `L2DebtManager` allows liquidators to liquidate loans that pass the liquidation threshold. The liquidator has to repay the loan with the same tokes as was borrowed and, in return, receives the borrower's collateral. However, there is no bonus or discount for the liquidator, meaning that during the liquidation, the amount of collateral is equivalent to the repaid borrow:

```
function liquidate(
    address user,
    address borrowToken,
    uint256 debtAmountInUsdc
) external {
    ...
    // @audit: amount in debtAmountInUsdc is
    //         repaying the debt
    IERC20(borrowToken).safeTransferFrom(
        msg.sender,
        address(this),
        _convertFromSixDecimals(borrowToken, debtAmountInUsdc)
    );

    ...
    // @audit: the equivalent amount of collateral
    //         is calculated
    TokenData[] memory collateralTokensToSend =
    _getCollateralTokensForDebtAmount(
            user,
            debtAmountInUsdc
        );

    uint256 len = collateralTokensToSend.length;

    for (uint256 i = 0; i < len; ) {
        ...
        // @audit: calculated collateral tokens are
        //         transferred to the liquidator
        IERC20(collateralTokensToSend[i].token).safeTransfer(
            msg.sender,
            collateralTokensToSend[i].amount
        );

        unchecked {
            ++i;
        }
    }
    ...
}
```

```solidity
// @audit: this function calculates amount of
//         collateral that should be sent to the
//         liquidator for the repaid loan
function _getCollateralTokensForDebtAmount(
    address user,
    uint256 repayDebtUsdcAmt
) internal view returns (TokenData[] memory) {
    uint256 len = _supportedCollateralTokens.length;
    TokenData[] memory collateral = new TokenData[](len);

    for (uint256 i = 0; i < len; ) {
        address collateralToken = _supportedCollateralTokens[i];
        // @audit: Iterate over each supported collateral token and
        //         calculate the current collateral value
        uint256 collateralAmountForDebt = convertUsdcToCollateralToken(
            collateralToken,
            repayDebtUsdcAmt
        );
    ...
    }

    return collateral;
}

// @audit: function to calculate current value
//         of the collateral token.
function convertUsdcToCollateralToken(
    address collateralToken,
    uint256 debtUsdcAmount
) public view returns (uint256) {
    if (!isCollateralToken(collateralToken))
        revert UnsupportedCollateralToken();
    return
    // @audit: the value is calcualated
    //         but there is no bonus or discount accounted
        (debtUsdcAmount * 10 ** _getDecimals(collateralToken)) /
        IPriceProvider(_cashDataProvider.priceProvider()).price(
            collateralToken
        );
}
```

As can be seen, there is no accounting for bonuses or discounts for the liquidator. Lack of incentivization for the liquidators results in inefficient liquidations and a high risk of bad debt creation, which causes loss for the borrow tokens provider.

Recommendation(s): Consider rewarding liquidators for liquidating by sending a bonus amount of collateral.

Status: Fixed

Update from the client: Each token has a liquidation bonus.

Update from the Nethermind Security: The `liquidationBonus` in new structure `LiquidationTokenData` is not used.

6.8 [Medium] Features that are planned to add require substantial changes in the code

File(s): `L2DebtManager.sol`

Description: The team informed that some features are planned to be added or changed. However those would require changing the existing code:

- liquidation is now permissioned, but is planned to be permissionless;
- other tokens than weETH are planned to be used as collateral, but that would require creating a new `PriceProvider` contract;

Recommendation(s): Consider creating more general logic which would allow make changes to the protocol by setting parameters instead of performing code changes.

Status: Acknowledged

Update from the client: With the recent changes, liquidations have become permissionless. Price provider will change whenever a new token is added.

6.9 [Medium] Incorrect calculation in `_getAmountWithInterest(...)`

File(s): `L2DebtManager.sol`

Description: The information provided by the client points that the value in `_borrowApyPerSecond` defines interest per second expressed in basis points (with `18` decimals). However, during the calculation of the interest in `_getAmountWithInterest(...)`, the `_borrowApyPerSecond` is treated as a percentage, since it is divided by the `100` (actually by `1e20`, but `1e20/1e18=100`):

```
function _getAmountWithInterest(
    uint256 amountBefore,
    uint256 accInterestAlreadyAdded
) internal view returns (uint256) {
    return
        ((1e18 *
            (amountBefore *
                (debtInterestIndexSnapshot() - accInterestAlreadyAdded))) /
         // @audit: division by 100 treats the _borrowApyPerSecond
         // like percentage, not basis points
            1e20 +
            1e18 *
            amountBefore) / 1e18;
}
```

Since the basis points are expressed as `1/100th` of `1%`, the division should be done by `1e22`. If the `_borrowApyPerSecond` is defined in basis points, the above calculation would be incorrect, and the increase of interest would be a hundred times faster.

Recommendation(s): Consdier clearly defining and documenting the unit of `_borrowApyPerSecond`.

Status: Mitigated

Update from the client: The `borrowApyPerSecond` has 18 decimals and is expressed in percentage

6.10 [Medium] Lack of collateral cap

File(s): `L2DebtManager`

Description: Currently, there is no maximum amount of the collateral token that the users can deposit. That is unsafe, as if single asset is used in high volumes for the loans, a drop in the price may cause many loans enter unhealthy state. Further liquidations would result in the immediate selling of the collateral asset, resulting in a collateral token price drop even more, possibly causing new liquidations, causing the effect known as cascading liquidations.

Recommendation(s): To mitigate this issue, there can be a set cap for each collateral token. This would distribute the collateral pool and protect from the described scenario.

Status: Fixed

Update from the client: Added a supply cap per collateral token.

6.11 [Medium] Lack of collateral control on liquidation

File(s): `L2DebtManager`

Description: The `L2DebtManager` controls only which tokens are supported as collateral. However, there is no control of which collateral tokens constitue most of the pool. There is no mechanism that would allow to define different liquidation bonuses for each collateral token. Moreover, the liquidators can't choose which token they would receive for the debt repayment. Those mechanisms are important as they allow to constitute the majority of the collateral as less volatile assets by incentivizing liquidators to choose more volatile assets with higher bonuses. In the current design, the collateral paid to the liquidator is defined by the order of support, so if `ETH` was the first supported token, then it will be a first collateral token that is sent to the liquidator. This causes additional problems because the asset that causes the liquidations may not be the one that is removed from the pool. Consider following scenario:

- admin supported two tokens as collateral, `WETH` and `DOGE`;
- the price of `DOGE` starts to drop quickly;
- liquidations start to occur, but first, the more stable asset `WETH` is removed from the `L2DebtManager` as that's the order of paying our to liquidators;
- the `L2DebtManager` is left with the more volatile asset `DOGE`, causing more liquidations;

Recommendation(s): Consider allowing liquidators to choose which collateral token they receive and implementing a mechanism that allows them to set each collateral token with a different liquidation bonus.

Status: Fixed

Update from the client: Added collateral token preference which the liquidator can pass as a function argument. Liquidation bonus for each collateral token was introduced as a fix to the "[High] Lack of incentive for liquidators" issue.

6.12 [Medium] Liquidations can be blocked

File(s): L2DebtManager

Description: If the L2DebtManager supports as collateral a token that reverts transfer with amount = 0, some loans may become impossible to liquidate. The liquidate(...) function iterates over the collateralTokensToSend array and transfers the token amount defined in TokenData.amount. But some of the elements may contain empty TokenData.amount. If one of the collateral tokens reverts on transfer of amount == 0, the liquidation would revert as well. As a result the unhealthy loan may not be possible to liquidate.

Recommendation(s): Consider mitigating the described issue by doing token transfer only to non-zero amounts.

Status: Fixed

Update from the client: Added a check where if the amount != 0, then only transfer funds.

6.13 [Medium] Optional partial liquidations

File(s): L2DebtManager

Description: Currently, the L2DebtManager allows for partial liquidations. However, the liquidator would not choose to liquidate only part of the loan if liquidating the whole loan would result in a higher profit. Therefore, we can assume that only full liquidations would practically happen. This is not attractive for the borrowers, as if their loan becomes unhealthy, the whole loan is liquidated. Partial liquidations allow the users to lose less money, as after the partial liquidation, the loan may enter a healthy state again. Additionally, this particularly disincentivizes large loans, as with a larger amount of borrowed assets, the potential loss at full liquidation is greater. Moreover, full liquidation on large loans introduces risk to the protocol. If the liquidation of large loan is motivated by decreasing value of the collateral asset, the liquidator is most likely to sell the collateral immediately after receving it on liqudiation. This may affect the price of the asset and result in putting other loans in an unhealthy state.

Recommendation(s): Consider setting partial liquidation as a default (e.g. 50%) and only allow for full liquidation if after the partial liquidation loan does no get into a healthy state. Additionally, consider applying a dynamic liquidation threshold depending on the size of a loan, which would allow for a lower price impact on liquidation.

Status: Fixed

Update from the client: Added 50% liquidation initially, then if user is still liquidatable, a 100% liquidation will be triggered in the same function.

Update from the Nethermind Security: The applied changes apply liquidation of half of the loan. However, the dynamic percentage of liquidations depending on loan size is not introduced.

6.14 [Medium] Price Oracle may not support collateral assets

File(s): UserSafe.sol

Description: Collateral assets are added with the addCollateral(...) and addCollateralAndBorrow(...) functions. However, when adding new tokens, there are no checks to ensure that the PriceProvider contract supports those tokens. If such a token is supported and the user deposits it as the collateral, then during price calculation in convertCollateralTokenToUsdc(...) and convertUsdcToCollateralToken(...), the call to PriceProvider may revert. In this case, whole user's collateral is unusable, until user withdraws the incorrect token from the debt manager. Additionally, currently, the PriceProvider supports only the weETH token:

```
function price(address token) external view returns (uint256) {
    if (token != weETH) revert UnknownToken();
    ...
}
```

Recommendation(s): Consider checking if collateral token is supported by the PriceProvider when adding collateral tokens to the contract.

Status: Fixed

Update from the client: Added a check on price in support collateral function.

6.15 [Medium] Unhandled bad debt

File(s): L2DebtManager

Description: The L2DebtManager does not have a robust mechanism to remove bad debt. Currently, a loan that creates bad debt because the debt value is higher than the value of the collateral can't be liquidated. This happens because the liquidate(...) function requires the loan to be healthy after the repayment:

```
function liquidate(
    address user,
    address borrowToken,
    uint256 debtAmountInUsdc
) external {
    _updateBorrowings(user, borrowToken);
    // @audit: check if user is liquidatable
    if (!liquidatable(user)) revert CannotLiquidateYet();
    // @audit: perform liquidation
    ...
    // @audit: to finish liquidation the loan must be in repaid, non-liquidatable state
    if (liquidatable(user))
        revert PartialLiquidationShouldOverCollaterallizeTheUser();
    ...
}
```

This requirement does not allow the liquidators to liquidate unhealthy loan, even though the `liquidatable(...)` function shows that the loan can be liquidated. Such situation could be resolved by repaying part of the loan so that it can be liquidatable, but this could be not profitable for the liquidators. If the EtherFi Cash team decides to do it, it will be a loss for the protocol.

Recommendation(s): A common solution for dealing with bad debt is to socialize the loss, meaning it is spread by all the borrowing token suppliers. Such a solution allows for lowering the impact of the bad debt. Note that alternatively, this problem may be solved by EtherFi team covering the costs of bad debt by repaying enough debt so that the loan can be liquidated. However, this requires constant monitoring of all the loans.

Status: Mitigated

Update from the client: We've added a possibility to liquidate loan with a bad debt up to a collateral value. Additionaly, we plan to monitor all the users and in case of bad debt EtherFi team would liquidate and repay such loan. Moreover, we expect low probability of bad debt as all users are KYC'ed and in case of not repaying loan with a bad deby the user may face legal consequences.

6.16 [Medium] Unsupporting tokens may be practically impossible

File(s): `L2DebtManager.sol`

Description: The collateral tokens may become unsupported by the admin with the `unsupportCollateralToken(...)` function. However, the process of the unsupporting token may be practically difficult to perform as it requires every user user to withdraw that token from the `L2DebtManager` contract:

```
function unsupportCollateralToken(
    address token
) external onlyRole(ADMIN_ROLE) {
    if (token == address(0)) revert InvalidValue();
    // @audit: No user can hold this token to unsupport it
    if (_totalCollateralAmounts[token] != 0)
        revert TotalCollateralAmountNotZero();
    ...
}
```

If the token needs to be unsupported, if any user holds it in the debt manager, the admin can't unsupport that token, and users can still deposit it. Note that a malicious actor may intentionally not withdraw such a token, therefore blocking the unsupport mechanism.

Recommendation(s):

Status: Acknowledged

Update from the client: In an extreme case of unsupporting tokens, we plan to modify the `L2DebtManager` contract and introduce the migration logic.

6.17 [Medium] Use of deprecated Chainlink Function leading to insufficient data feed validation

File(s): `PriceProvider.sol`

Description: Chainlink as seen here deprecated the use of `latestAnswer`. The use of `latestRoundData()` is advised in turn because it allows for information validation such as timestamp. In the `PriceProvider` contract, the `price(...)` function is used which shows the use `latestAnswer()` as seen below:

```
function price(address token) external view returns (uint256) {
    if (token != weETH) revert UnknownToken();

    int256 priceWeEthWeth = weEthWethOracle.latestAnswer();
    if (priceWeEthWeth <= 0) revert PriceCannotBeZeroOrNegative();
    int256 priceEthUsd = ethUsdcOracle.latestAnswer();
    if (priceEthUsd <= 0) revert PriceCannotBeZeroOrNegative();

    uint256 returnPrice = (uint256(priceWeEthWeth) *
        uint256(priceEthUsd) *
        10 ** DECIMALS) / 10 ** (weEthWethDecimals + ethUsdcDecimals);

    return returnPrice;
}
```

As a result, stale prices can be used in the contract because `latestAnswer()` does not perform sdata freshness checks.

Recommendation(s): Consider using `latestRoundData()` instead of `latestAnswer()` in the `PriceProvider` function.

Status: Mitigated

Update from the client: We won't be using this price provider.

6.18 [Medium] `L2DebtManger` is permissionless

File(s): `L2DebtManager.sol`

Description: The functions of the `L2DebtManager` contract are permissionless, so any user could use the debt manager to, e.g., supply a borrow token or take a loan. However, the documentation provided by the client points out that only liquidation should be permissionless, and other functionalities should be available only from the `UserSafe` contracts.

Recommendation(s): Consider restricting functions that should be accessible only by the `UserSafe`.

Status: Fixed

Update from the client: Supply/Withdrawal of the borrow tokens, repayments of borrow tokens and liquidations stay permissionless.

Update from the Nethermind Security: Only `UserSafe` contracts are able to deposit collateral. However, it is possible to provide collateral to the address that is not `UserSafe` contract. Consider checking on deposit if the `user` is whitelisted `UserSafe` contract.

6.19 [Low] Updating borrowings with collateral token

File(s): `L2DebtManager.sol`

Description: The `withdrawCollateral(...)` function calls `_updateBorrowings(...)` with a provided collateral token. This action is incorrect since there is no need to update debt interest on the collateral token.

Recommendation(s): Consider removing `_updateBorrowings(...)` from `withdrawCollateral(...)` function.

Status: Fixed

6.20 [Low] Any tokens can be supplied as borrow tokens

File(s): `L2DebtManager`

Description: The `supply(...)` function does not validate if the provided token is a borrow token. This allows to supply any token and receive shares for that token.

```solidity
function supply(
    address user,
    address borrowToken,
    uint256 amount
) external {
    // @audit: no checks if the token is a borrow token
    uint256 shares = _convertBorrowToShare(
        borrowToken,
        (_borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens == 0)
            ? amount
            : (amount *
                _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens) /
                _getTotalBorrowTokenAmount(borrowToken)
    );

    IERC20(borrowToken).safeTransferFrom(msg.sender, address(this), amount);

    _sharesOfBorrowTokens[user][borrowToken] += shares;
    _borrowTokenConfig[borrowToken].totalSharesOfBorrowTokens += shares;

    emit Supplied(msg.sender, user, borrowToken, amount);
}
```

Recommendation(s): Consider applying check for borrow token in the `supply(...)` function.

Status: Fixed

Update from the client:

6.21 [Low] Lack of `disableInitializers()`

File(s): `UserSafe.sol`, `L2DebtManager.sol`

Description: The `_disableInitializers()` function prevents the implementation contract from being reinitialized once deployed. Both the `UserSafe` and the `L2DebtManager` contracts lack this function in their respective constructors. More information on using the `disableInitializers()` can be found here: https://docs.openzeppelin.com/contracts/4.x/api/proxyInitializable-_disableInitializers–

Recommendation(s): Consider adding the `_disableInitializers()` in the constructor of both contracts

Status: Fixed

Update from the Nethermind: Resolved by changes in commit `d97d9a9`.

Update from the client:

6.22 [Low] Lack of minimum borrow amounts

File(s): `L2DebtManager`

Description: The borrowers can lend any amount of the borrow tokens. If the borrower would lend very small amount then even if the loan becomes liquidatable no liquidator would be interested to perform liquidation, as gas costs may be higher than the gained rewards (note that current rewards are `0`). The malicious borrower could create many small loans that, even in a liquidatable state, would potentially not be liquidated, creating bad debt.

Recommendation(s): Consider setting a minimum amount of the loan.

Status: Acknowledged

Update from the client: We don't think this has any effect on the protocol. That kind of borrowing will likely consume more gas than total being borrowed, so there is no incentive.

Update from the Nethermind Security: Note that the small amount of debt may be as well created by not repaying the full loan, leaving in the debt manager small amount of debt. This debt may accrue interest and brings risk of creating bad debt which would not be handled by external liquidators, due to small size.

6.23 [Low] Not following the CEI pattern

File(s): `L2DebtManager.sol`

Description: The checks, effects and interaction pattern ensures that all required checks are done before updating the state of the contract, then the contract can make external function calls. This pattern is mitigates against re-entrancy issues.

The `depositCollateral(...)` function in `L2DebtManager` performs an external transfer call and then updates the state of the contract. Additionally, this function does not have a non-reentrant modifier to prevent reentrancy attacks.

```
function depositCollateral(
        address token,
        address user,
        uint256 amount
    ) external updateBorrowings(user) {
        if (!isCollateralToken(token)) revert UnsupportedCollateralToken();

        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
        _totalCollateralAmounts[token] += amount;
        _userCollateral[user][token] += amount;

        emit DepositedCollateral(msg.sender, user, token, amount);
    }
```

Recommendation(s): Consider implementing checks, effects and interaction pattern and perform the external transfer function last.

Status: Fixed

Update from Nethermind Security: There are still functions that do not follow CEI pattern or did not implement `nonReentrant` modifier: `supply(. . . )`, `closeAccount(. . . )`, `_repayWithBorrowToken(. . . )` (in `repay(...)`).

Update from the client: Fixed at 867b3b4c66ea715079700b9cf07891c13089d852 in `DebtManagerCore`.

6.24 [Low] The `_requestWithdrawal(...)` function does not check for duplicates

File(s): `UserSafe.sol`

Description: The `UserSafe` contract does not account for duplicates when requesting withdrawals. Withdrawal of tokens in the `UserSafe` contract is done in two steps. First, the user calls `requestWithdrawal(...)`, and then after the waiting period, they can call `processWithdrawal(...)`. During the request for withdrawal, the contract checks if the contract has enough token balance for the user's withdrawal request, as shown below:

```
    function _requestWithdrawal(
        address[] calldata tokens,
        uint256[] calldata amounts,
        address recipient
    ) internal {
        _cancelOldWithdrawal();

        uint256 len = tokens.length;
        if (len != amounts.length) revert ArrayLengthMismatch();

        uint96 finalTime = uint96(block.timestamp) + _cashDataProvider.delay();

        for (uint256 i = 0; i < len; ) {
            // @audit: innefecitve check since the same token may be provided twice in the array
            if (IERC20(tokens[i]).balanceOf(address(this)) < amounts[i])
                revert InsufficientBalance();

            unchecked {
                ++i;
            }
        }
```

If the same token was provided more than once, the balance check would pass for each iteration even if the sum of those tokens would not pass the check. During the transfer, the transaction would fail without a clear error message. The user would have to create another correct withdrawal request.

Recommendation(s): Consider implementing mechanism that would account for duplicates during balance checks.

Status: Acknowledged

Update from the client: Don't want to increase gas by putting this check if it does not harm the protocol.

6.25 [Low] The recovery mechanism does not protect the Safe owner

File(s): `UserSafe.sol, UserSafeRecovery.sol`

Description: The `UserSafe` contract allows for the user's safe recovery through the `recoverUserSafe(...)` function.

```
    function recoverUserSafe(
        bytes calldata newOwner,
        Signature[2] calldata signatures
    ) external onlyWhenRecoveryActive incrementNonce {
        _recoverUserSafe(_nonce, signatures, newOwner);
    }
```

This immediately recovers user's safe and transfers safe ownership to a new address. The `UserSafe's` ownership is a sensitive process and should occur in a two step process with a waiting period during which the current owner could revoke the ownership transfer, to mitigation malicious safe recovery.

Recommendation(s): Consider applying mechanisms that would protect the safe owner from malicious ownership transfer.

Status: Fixed

Update from the client: When recovery is called, an incoming owner is set. This owner comes into effect after a delay. The owner can cancel the incoming owner by calling the setOwner function within the delay.

6.26 [Info] No distinction between the KYC and non-KYC `UserSafe` contracts

File(s): `UserSafeFactory.sol`, `UserSafe.sol`

Description: Any address may create a new `UserSafe` contract by calling `UserSafeFactory`. However, the owners of the `UserSafe` should only be the addresses of the users that went through the EtherFi's KYC process.

Recommendation(s): While this issue may be resolved with the off-chain infrastructure that will only consider approved addresses of the `UserSafe`, consider adding on-chain checks and verification.

Status: Fixed

Update from the client: The `createUserSafe(...)` is callable only by the admin role.

6.27 [Info] No checks of liquidation threshold compared to liquidation bonus

File(s): `L2DebtManager`

Description: There are no checks to determine whether the liquidation bonuses can be covered by the buffer set by the liquidation threshold. If the parameters are set incorrectly, the liquidation may not be liquidatable because of the high bonus that is not covered by the collateral.

Recommendation(s): Consider carefully setting the parameters of the liquidation threshold and liquidation bonus so to lower the risk of bad debt.

Status: Unresolved

6.28 [Info] Non-USD stablecoins can't be supported

File(s): `L2DebtManager.sol`

Description: EtherFi Cash protocol allows users to borrow stablecoins. However, current design only allows for borrowing only stablecoins pegged to USD. Supporting other stablecoins would require protocol upgrade.

Recommendation(s): Consider defining what stablecoins are planned to be used as borrowed assets.

Status: Mitigated

Update from the client: Current goal is to only support USD stablecoins for debt positions.

6.29 [Info] The `CREATE3` library can't be used on zkSync Era

File(s): `UserSafeFactory.sol`

Description: The `UserSafeFactory` contract uses the `CREATE3` mechanism to deploy `UserSafe` contracts. This allows to keep the same address of the deployed contracts over different `L2` chains. However, The used `CREATE3` library can't be used on `ZKsync`, since the proxy in `CREATE3` is written directly in EVM bytecode, which is different from the bytecode that zkEVM operates on (docs). As a result, the library would have to be modified by adding the zkSync Era compatible bytecode to make it work. Moreover, the deployed contract address would differ from the addresses of `UserSafe` deployed on other L2s.

Recommendation(s): Consider important differences between Ethereum and zkSync Era before deployment.

Status: Acknowledged

Update from the client: There are no plans to deploy on the zkSync Era.

6.30 [Info] The `accInterestAlreadyStored` variable is unused

File(s): `L2DebtManager.sol`

Description: The modifier `updateBorrowings(...)` has a variable called `accInterestAlreadyStored`, which is updated by `_debtInterestIndexSnapshot` as can be seen here:

```
uint256 accInterestAlreadyStored = _debtInterestIndexSnapshot;
```

However, this variable is not used anywhere in the contract.

Recommendation(s): Consider removing unused variable in the code.

Status: Fixed

Update from the Nethermind: Resolved by changes in commit `d97d9a9`.

Update from the client:

6.31 [Info] The `onlyOwner` is usable only if the owner is an ETH address

File(s): `UserSafe.sol`

Description: The `UserSafe` contract has an `onlyOwner` modifier which allows only the owner to do certain critical operations. This modifier is implemented as follows:

```
    modifier onlyOwner() {
        _ownerBytes._onlyOwner();
        _;
    }
```

The `OwnerLib` implements the function as:

```
    function _onlyOwner(bytes memory _ownerBytes) internal view {
        if (_ownerBytes.length != 32) revert OnlyOwner();

        address __owner;
        assembly ("memory-safe") {
            __owner := mload(add(_ownerBytes, 32))
        }

        if (msg.sender != __owner) revert OnlyOwner();
    }
```

Therefore, if the owner variable does not hold an ETH Address, the contract reverts. Therefore, the `onlyOwner` modifier can be used only for the Ethereum addresses. According to the team, the owner could potentially be also defined by other public key.

Recommendation(s): Consider clearly defining ownership over the user safe.

Status: Fixed

Update from the client: Removed the modifier.

6.32 [Info] The recovery pubkeys may only be ETH addresses

File(s): `UserSafeRecovery.sol`

Description: The pubkeys provided to the `UserSafeRecovery` constructor currently can only be Ethereum addresses. However, the team plans to allow other signature mechanisms than signatures created with ETH address for the recovery mechanism.

Recommendation(s): Consider generalizing the recovery mechanism participants.

Status: Acknowledged

Update from the client: It is meant to be only ETH addresses.

6.33 [Info] Unclear result in `debtRatioOf(...)`

File(s): `L2DebtManager`

Description: The function `debtRatioOf(...)` returns the debt ratio of a loan:

```
function debtRatioOf(address user) public view returns (uint256) {
    (, uint256 totalDebtValue) = borrowingOf(user);
    uint256 collateralValue = getCollateralValueInUsdc(user);
    if (collateralValue == 0) revert ZeroCollateralValue();

    return (totalDebtValue * 1e20) / collateralValue; // result in basis points
}
```

However, since the `totalDebtValue` and `collateralValue` are in six decimals, the result of a ratio would be in `18` decimals. This may be confusing to users and liquidators since view functions like `borrowingOf(...)` or `collateralOf(...)` return results in the `USDC` value in six decimals.

Recommendation(s): Consider clearly documenting the value returned in `debtRatioOf(...)`.

Status: Fixed

Update from the client: Removed the function.

6.34 [Info] Unnecessary owner in the recovery mechanism

File(s): `UserSafeRecovery.sol`

Description: The `recoverUserSafe(...)` function requires two recovery signatures to be able to recover a user safe account. However, the `owner` is added to checks in the `_getRecoveryOwner(...)` function as follows:

```
    function _getRecoveryOwner(
        uint8 index
    ) internal view returns (OwnerLib.OwnerObject memory) {
        if (index == 0) return this.owner();
        else if (index == 1) return _etherFiRecoverySigner.getOwnerObject();
        else if (index == 2) return _thirdPartyRecoverySigner.getOwnerObject();
        else revert InvalidSignatureIndex();
    }

```

The owner here is unneccessary, since the recovery mechanism is meant to be used in case where the current owner lost their account access.

Recommendation(s): Consider removing current owner from the owner recovery mechanism. If the owner should be able to transfer ownership, consider implementing separate functionality that would allow for this.

Status: Fixed

Update from the client: Added a user recovery signer which can be set by the user. Now, there will be a total of 3 recovery signers. User recovery signer set by the user (their ledger etc). Etherfi recovery signer Third party recovery signer. Quorum is still 2/3.

6.35 [Info] Unused Code

File(s): `L2DebtManager.sol`

Description: The `L2DebtManager.sol` contract has a constant `uint256 public constant AN_YEAR_IN_SECONDS = 365 * 24 * 60 * 60;` which is not used anywhere in the contract.

Recommendation(s): Consider removing unused code in the contract to improve code clarity and code hygiene.

Status: Fixed

Update from the client: Removed the variable.

6.36 [Info] Unused comparison expressions

File(s): `L2DebtManager.sol`

Description: The functions `unsupportCollateralToken(...)` and `unsupportBorrowToken(...)` contain unused comparison expressions:

```
_collateralTokenIndexPlusOne[_supportedCollateralTokens[len - 1]] ==
            indexPlusOneForTokenToBeRemoved;
```

and

```
_borrowTokenIndexPlusOne[_supportedBorrowTokens[len - 1]] ==
            indexPlusOneForTokenToBeRemoved;
```

Recommendation(s): Consider removing unused code to improve readability.

Status: Fixed

6.37 [Best Practice] Improve code clarity and readability by using constants instead of magic numbers.

File(s): L2DebtManager.sol

Description: The code contains multiple instances of magic numbers, which are literal values used directly in the code without explanation. These magic numbers can make the code harder to read and maintain. Specifically, there are instances where magic numbers are used, such as `10`, `1e6`, `1e20`, `1e18`. Using magic numbers can lead to errors and makes the code less flexible.

Recommendation(s):

To improve code clarity and maintainability, replace these magic numbers with well-named constants that describe their purpose. This approach will make the code more understandable and easier to update in the future. Suggested constants could be:

`uint256 public constant PRECISION = 1e18 uint256 public constant PRECISION_PERCENTS = 1e20 uint256 public constant SIX_DECIMALS = 1e6`

For example, the debt ratio calculation in `debtRatioOf` could be rewritten as:

```
return (debtValue * PRECISION_EXTENDED) / collateralValue;
```

Similarly the `_getAmountWithInterest` gets much more readeable after using the suggested constants:

```
function _getAmountWithInterest(
    uint256 amountBefore,
    uint256 accInterestAlreadyAdded
) internal view returns (uint256) {
    return
        ((PRECISION *
            (amountBefore *
                (debtInterestIndexSnapshot() - accInterestAlreadyAdded))) /
            PRECISION_EXTENDED +
            PRECISION *
            amountBefore) / PRECISION;
}
```

Status: Fixed

6.38 [Best Practice] Use AccessControlDefaultAdminRulesUpgradeable

File(s): L2DebtManager.sol

Description: The `L2DebtManager` contract uses OpenZeppelin's `AccessControlUpgradable` to manage access control within the contract. But this is not advisable by Openzeppelin, as can be seen in this recommendation, where the OpenZeppelin team advises to use `AccessControlDefaultAdminRulesUpgradeable` which provides additional security.

Recommendation(s): Consider using `AccessControlDefaultAdminRulesUpgradeable` instead of `AccessControlUpgradable`.

Status: Fixed

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the EtherFi Cash documentation

The documentation for the EtherFi Cash protocol was provided by the team during kick-off meeting. Additional documentation in the form of diagrams.
The team answered every question during meetings or through messages, which gave the auditing team a lot of insight and a deep understanding of the technical aspects of the project.

8 Test Suite Evaluation

```
% forge test
[] Compiling...
[] Compiling 189 files with 0.8.24
[] Solc 0.8.24 finished in 36.23s

Ran 1 test for test/DebtManager/Deploy.t.sol:DebtManagerDeployTest
[PASS] test_Deploy() (gas: 81654)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.47ms (333.21µs CPU time)

Ran 2 tests for test/AaveAdapter/AaveAdapter.t.sol:AaveAdapterTest
[PASS] test_Flow() (gas: 131984)
[PASS] test_Process() (gas: 144090)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.76ms (620.63µs CPU time)

Ran 1 test for test/DebtManager/CloseAccount.t.sol:DebtManagerCloseAccountTest
[PASS] test_CloseAccount() (gas: 110099)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.21ms (1.30ms CPU time)

Ran 2 tests for test/UserSafe/Deploy.t.sol:UserSafeDeployTest
[PASS] test_Deploy() (gas: 66492)
[PASS] test_DeployAUserSafe() (gas: 306734)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.49ms (205.63µs CPU time)

Ran 5 tests for test/DebtManager/Liquidate.t.sol:DebtManagerLiquidateTest
[PASS] test_CannotLiquidateIfNotLiquidatable() (gas: 120006)
[PASS] test_Liquidate() (gas: 201526)
[PASS] test_OnlyAdminCanSetLiquidationThreshold() (gas: 24701)
[PASS] test_PartialLiquidate() (gas: 211195)
[PASS] test_SetLiquidationThreshold() (gas: 35331)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 10.16ms (2.29ms CPU time)

Ran 1 test for test/Swapper/Swapper1Inch.t.sol:Swapper1InchV6Test
[PASS] test_Swap() (gas: 70639)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.35ms (77.25µs CPU time)

Ran 13 tests for test/DebtManager/Collateral.t.sol:DebtManagerCollateralTest
[PASS] test_CanAddOrRemoveSupportedCollateralTokens() (gas: 142354)
[PASS] test_CannotAddCollateralTokenIfAlreadySupported() (gas: 22988)
[PASS] test_CannotAddNullAddressAsCollateralToken() (gas: 18618)
[PASS] test_CannotDepositCollateralIfAllownaceIsInsufficient() (gas: 213788)
[PASS] test_CannotDepositCollateralIfBalanceIsInsufficient() (gas: 63120)
[PASS] test_CannotDepositCollateralIfTokenNotSupported() (gas: 19885)
[PASS] test_CannotUnsupportAddressZeroAsCollateralToken() (gas: 18559)
[PASS] test_CannotUnsupportAllTokensAsCollateral() (gas: 27182)
[PASS] test_CannotUnsupportCollateralTokenIfTotalCollateralNotZeroForTheToken() (gas: 118561)
[PASS] test_CannotUnsupportTokenForCollateralIfItIsNotACollateralTokenAlready() (gas: 25088)
[PASS] test_DepositCollateral() (gas: 184783)
[PASS] test_LtvCannotBeGreaterThanLiquidationThreshold() (gas: 20691)
[PASS] test_OnlyAdminCanSupportOrUnsupportCollateral() (gas: 30620)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 10.98ms (3.69ms CPU time)

Ran 6 tests for test/IntegrationTest/IntegrationTest.t.sol:IntegrationTest
[PASS] test_AddCollateral() (gas: 332504)
[PASS] test_AddCollateralAndBorrow() (gas: 472688)
[PASS] test_CloseAccount() (gas: 610196)
[PASS] test_MultipleSuppliers() (gas: 2452949)
[PASS] test_RepayUsingUsdc() (gas: 653208)
[PASS] test_WithdrawCollateral() (gas: 517520)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 11.81ms (4.28ms CPU time)

Ran 1 test for test/Swapper/SwapperOpenOcean.t.sol:SwapperOpenOceanTest
[PASS] test_Swap() (gas: 70639)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.18ms (61.67µs CPU time)
```

```
Ran 3 tests for test/DebtManager/FundManagement.t.sol:DebtManagerFundManagementTest
[PASS] test_FundsManagementOnAave() (gas: 182297)
[PASS] test_OnlyAdminCanManageFunds() (gas: 25298)
[PASS] test_supplier() (gas: 707429)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 2.56ms (1.04ms CPU time)

Ran 5 tests for test/DebtManager/Repay.t.sol:DebtManagerRepayTest
[PASS] test_CanRepayForOtherUser() (gas: 247492)
[PASS] test_CannotRepayWithUsdcIfAllowanceIsInsufficient() (gas: 56751)
[PASS] test_CannotRepayWithUsdcIfBalanceIsInsufficient() (gas: 157846)
[PASS] test_RepayAfterSomeTimeIncursInterestOnTheBorrowings() (gas: 127388)
[PASS] test_RepayWithUsdc() (gas: 77520)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 2.69ms (787.67µs CPU time)

Ran 2 tests for test/UserSafe/CollateralLimit.t.sol:UserSafeCollateralLimitTest
[PASS] test_CannotAddMoreCollateralThanCollateralLimit() (gas: 229224)
[PASS] test_SetCollateralLimit() (gas: 128430)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.65ms (794.38µs CPU time)

Ran 2 tests for test/UserSafe/Owner.t.sol:UserSafeOwnerTest
[PASS] test_CanSetEthereumAddrAsOwner() (gas: 77607)
[PASS] test_CanSetPasskeyAsOwner() (gas: 102123)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.07ms (1.00ms CPU time)

Ran 2 tests for test/UserSafe/UserSafeFactory.t.sol:UserSafeFactoryTest
[PASS] test_Deploy() (gas: 38146)
[PASS] test_Upgrade() (gas: 36513)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.70ms (97.71µs CPU time)

Ran 1 test for test/PriceProvider/PriceProvider.t.sol:PriceProviderTest
[PASS] test_Value() (gas: 12505)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.74ms (28.75µs CPU time)

Ran 2 tests for test/UserSafe/ReceiveFunds.t.sol:UserSafeReceiveFundsTest
[PASS] test_ReceiveFunds() (gas: 64463)
[PASS] test_ReceiveFundsWithPermit() (gas: 2698)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.97ms (82.21µs CPU time)

Ran 15 tests for test/DebtManager/Borrow.t.sol:DebtManagerBorrowTest
[PASS] test_Borrow() (gas: 245288)
[PASS] test_BorrowIncursInterestWithTime() (gas: 182420)
[PASS] test_BorrowTokenWithDecimalsOtherThanSix() (gas: 1041502)
[PASS] test_CanAddOrRemoveSupportedBorrowTokens() (gas: 688535)
[PASS] test_CannotAddBorrowTokenIfAlreadySupported() (gas: 22879)
[PASS] test_CannotAddNullAddressAsBorrowToken() (gas: 18620)
[PASS] test_CannotBorrowIfDebtRatioGreaterThanThreshold() (gas: 189928)
[PASS] test_CannotBorrowIfNoCollateral() (gas: 125081)
[PASS] test_CannotBorrowIfTokenIsNotSupported() (gas: 17713)
[PASS] test_CannotBorrowIfUsdcBalanceInsufficientInDebtManager() (gas: 196802)
[PASS] test_CannotRemoveSupportIfBorrowTokenIsStillInTheSystem() (gas: 38095)
[PASS] test_CannotUnsupportAllTokensAsBorrowTokens() (gas: 126506)
[PASS] test_CannotUnsupportTokenForBorrowIfItIsNotABorrowTokenAlready() (gas: 22815)
[PASS] test_NextBorrowAutomaticallyAddsInterestToThePreviousBorrows() (gas: 259275)
[PASS] test_OnlyAdminCanSupportOrUnsupportBorrowTokens() (gas: 642225)
Suite result: ok. 15 passed; 0 failed; 0 skipped; finished in 5.19ms (3.30ms CPU time)

Ran 4 tests for test/UserSafe/SpendingLimit.t.sol:UserSafeSpendingLimitTest
[PASS] test_CannotSpendMoreThanSpendingLimit() (gas: 99588)
[PASS] test_SetSpendingLimit() (gas: 321786)
[PASS] test_SpendingLimitGetsRenewedAutomatically() (gas: 329120)
[PASS] test_UpdateSpendingLimitWithPermit() (gas: 305647)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 6.50ms (4.33ms CPU time)

Ran 3 tests for test/DebtManager/Withdraw.t.sol:DebtManagerWithdrawTest
[PASS] test_CannotWithdrawIfDebtRatioBecomesUnhealthyAfterWithdrawal() (gas: 117290)
[PASS] test_CannotWithdrawIfNotACollateralToken() (gas: 38392)
[PASS] test_Withdraw() (gas: 163311)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 2.55ms (301.50µs CPU time)

Ran 2 tests for test/UserSafe/WebAuthn.t.sol:UserSafeWebAuthnSignatureTest
[PASS] test_CanSetOwnerWithWebAuthn() (gas: 304031)
[PASS] test_Deploy() (gas: 74799)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 4.90ms (2.30ms CPU time)
```

```
Ran 6 tests for test/UserSafe/Withdrawal.t.sol:UserSafeWithdrawalTest
[PASS] test_CanResetWithdrawalWithNewRequest() (gas: 374674)
[PASS] test_CanTransferEvenIfAmountIsBlockedByWithdrawal() (gas: 324123)
[PASS] test_CannotProcessWithdrawalsBeforeTime() (gas: 315417)
[PASS] test_CannotRequestWithdrawalWhenFundsAreInsufficient() (gas: 126953)
[PASS] test_ProcessWithdrawals() (gas: 303634)
[PASS] test_RequestWithdrawalWithPermit() (gas: 332149)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 3.31ms (3.88ms CPU time)

Ran 6 tests for test/UserSafe/Recovery.t.sol:UserSafeRecoveryTest
[PASS] test_CanRecoverWithTwoAuthorizedSignatures() (gas: 287036)
[PASS] test_CanSetIsRecoveryActive() (gas: 63997)
[PASS] test_CannotRecoverIfRecoveryIsInactive() (gas: 72647)
[PASS] test_IsRecoveryActive() (gas: 18418)
[PASS] test_RecoveryFailsIfSignatureIndicesAreSame() (gas: 51592)
[PASS] test_RecoveryFailsIfSignatureIsInvalid() (gas: 69893)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 14.58ms (12.23ms CPU time)

Ran 11 tests for test/UserSafe/Transfers.t.sol:UserSafeTransfersTest
[PASS] test_AddCollateralToDebtManager() (gas: 175496)
[PASS] test_CannotAddCollateralIfBalanceIsInsufficient() (gas: 179165)
[PASS] test_CannotSwapAndTransferIfBalanceIsInsufficient() (gas: 74212)
[PASS] test_CannotTransferForSpendingWhenBalanceIsInsufficient() (gas: 208670)
[PASS] test_CannotTransferMoreThanSpendingLimit() (gas: 65692)
[PASS] test_CannotTransferUnsupportedTokensForCollateral() (gas: 45529)
[PASS] test_CannotTransferUnsupportedTokensForSpending() (gas: 44340)
[PASS] test_OnlyCashWalletCanTransferForSpending() (gas: 31264)
[PASS] test_OnlyCashWalletCanTransferFundsForCollateral() (gas: 31267)
[PASS] test_SwapAndTransferForSpending() (gas: 377112)
[PASS] test_TransferForSpendingToCashMultiSig() (gas: 132811)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 33.89ms (2.68ms CPU time)

Ran 9 tests for test/PreOrder.t.sol:PreOrderTest
[PASS] testAssemblyProperlySetsArrayLength() (gas: 19827)
[PASS] testMint() (gas: 4759012)
[PASS] testMintWithPermit() (gas: 3113094)
[PASS] testPause() (gas: 114544)
[PASS] testRevert() (gas: 2846624)
[PASS] testSellOutTiers() (gas: 3297785)
[PASS] testTiersLengthCheck() (gas: 15674103)
[PASS] testTokensForUser() (gas: 33305600)
[PASS] testURI() (gas: 37675)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 3.08s (3.10s CPU time)

Ran 2 tests for test/AaveUnitTest.t.sol:AaveUnitTest
[PASS] test_fullFlow() (gas: 848183)
[PASS] test_healthFactor() (gas: 523811)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 11.16s (16.10s CPU time)

Ran 25 test suites in 11.19s (14.39s CPU time): 107 tests passed, 0 failed, 0 skipped (107 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.